Selling train tickets by SMS

Steven Meyer

School of Computer and Communication Sciences

Semester Project

June 2010

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Khaled Ouafi
EPFL / LASEC

Abstract

Selling train tickets has evolved in the last ten years from queuing in the railway station, to buying tickets on the internet and printing them. Both alternatives are still viable options, though they are time consuming or need printing devices. Nowadays it is essential to offer a service that is as fast and efficient as possible: mobile phones provide an accessible, affordable and widely available tool for supplying information and transferring data.

The goal of this project is to design a train ticket contained in a SMS message. While there are several challenges related to the project, the main one is the security and how we can digitally sign a train ticket that is contained in 160 characters. The solution offered in this project is the implementation of the MOVA Signature (from the name of the inventors MOnnerat and VAudenay) that uses an interactive verification and therefore allows signature of 20 bits (roughly 4 characters).

Contents

Chapter 1

Introduction

In 1976, when Diffie and Hellman published their paper on public cryptosystem [6], they explained the concept of digital signature that corresponds for the electronic world to the handwritten signature. To implement a digital signature we use asymmetric cryptography with one private key and one public key. The private key used to decrypt or sign a message is different from the public one used to encrypt or verify, though both are related through a mathematical function; additionally, finding the private key on the basis of the public one must require complex calculations that cannot be realized in a reasonable time. In general the Signer uses his private key to sign the object and then anyone can use the public key to verify the validity of the signature. This system offers authentication by guaranteeing that the sender is the person he pretends to be and integrity by guaranteeing that the receiver gets what the sender has sent.

In 1989, Chaum and van Antwerpen proposed Undeniable Signatures [3] as a system that would not be universally verifiable (verifiable just by having the public key), but that would necessitate an interactive protocol between the Signer and the Verifier to be able to confirm or deny a signature.

In 2004, Monnerat and Vaudenay proposed an undeniable signature scheme named MOVA [9] that has the particularity to generate very short signatures while still maintaining a high security.

Switzerland is well known for its efficient railway service which is reliable and punctual though nor very fast. Tickets bought on board are taxed with a penalty fine, which leaves passengers with the options of queuing at the station, buying the tickets through internet and printing them, or receiving the tickets by MMS.

MMS is a protocol that allows to send/receive rich media up to several kilobytes on a cell phone and extends the SMS system which is limited to 140 bytes. The MMS service uses the data connection of the cellphone and therefore works only on modern phones with operators that support this protocol and of course is much more expensive (especially while roaming) than SMS.

Chapter 2

Preliminaries

2.1 Mathematical Background

To understand the rest of the paper, we will first have to define some mathematical concepts:

2.1.1 Algorithmic Complexity

Definition 1 (Negligible) *A function $f : \mathbb{N} \to \mathbb{R}^+$ is called negligible ($negl(\cdot)$) if for any polynomial $p : \mathbb{N} \to \mathbb{R}^+$ there exists an integer n_0 such that $n \geq n_0 \Rightarrow f(n) < \frac{1}{p(n)}$.*

Definition 2 (Efficient) *An algorithm is said to be efficient if its running time is bounded by a polynomial of the size of its inputs. Conversely, an operation is said to be efficient if it is carried by an efficient algorithm.*

2.1.2 Groups

A group $(G, *)$ is an algebraic structure defined as a set G together with a binary operation $*$: $G * G \to G$. We write "$a * b$" for the result of applying the operation $*$ on the two elements a and b of G. To have a group, $*$ must satisfy the following axioms:

- **Closure** $\forall a, b \in G : a * b \in G$.
- **Associativity** $\forall a, b, c \in G : (a * b) * c = a * (b * c)$.
- **Identity element** $\exists e \in G : \forall a \in G, e * a = a = a * e$.
- **Inverse element** $\forall a \in G, \exists b \in G : a * b = e = b * a$ (where e is the identity element).

2.1.3 Group Homomorphism

Given two groups $(G, *)$ and $(H, \cdot)$, a group homomorphism from $(G, *)$ to $(H, \cdot)$ is a function $h : G \to H$ such that $\forall a, b \in G : h(a * b) = h(a) \cdot h(b)$.

Definition 3 (Interpolation) *Given two groups $(G, *)$ and $(H, \cdot)$, and a subset $S := \{(x_1, y_1) \dots (x_s, y_s)\} \subseteq G \times H$, we say that the set of points S interpolates in a group homomorphism if there exists a group homomorphism $h : G \to H$ such that $h(x_i) = y_i, \forall i = 1, \dots, s$.*

Given $A \subseteq G \times H$ and $B \subseteq G \times H$, we say that A interpolates in a group homomorphism with B if $A \cup B$ interpolates in a group homomorphism.

n-S-Group Homomorphism Interpolation Problem

n-S-GHI problems uses two groups G and H, a set of points $S \subseteq G \times H$ and a positive integer n. Given $x_1, \dots, x_n$ chosen randomly, the Problem is to compute $y_1, \dots, y_n \in H$ such that $(x_1, y_1), \dots, (x_n, y_n)$ interpolates with S in a group homomorphism.

n-S-Group Homomorphism Interpolation Decisional Problem

n-S-GHID problem uses two groups G and H, a set of points $S \subseteq G \times H$ and a positive integer n. The problem is to *decide* if an instance I is generated by a set of points $(x_1, y_1), \dots, (x_n, y_n) \in X \times Y$ chosen at random such that it interpolates with S in a group homomorphism or is generated by randomly choosing n couples in $(G \times H)^n$.

Definition 4 (H-generation of G) *Let two groups G and H with $x_1, \dots, x_n \in G$ and $y_1, \dots, y_n \in H$, we say that $x_1, \dots, x_n$ H-generate of G if there exists at most one homomorphic function h such that*

$$\forall i = 1, \dots, n, h(x_i) = y_i$$

Definition 5 (Expert Group Knowledge) *Let G and H be two groups, let d be the cardinality of H, and the set $S = \{x_1, \dots, x_n\} \in G^n$, such that they H-generate G. We say an algorithm has an Expert Group Knowledge of G with the set S if it is able to find for a random $x \in G$ coefficients $a_1, \dots, a_n \in \mathbb{Z}_d$ and $r \in G$ such that*

$$x = dr + a_1 x_1 + \dots + a_s x_s$$

2.2 Cryptographic notions

We will now see some Cryptographic definitions and properties that will be used later on in the paper

2.2.1 Statistical Distances and Distinguishers

Definition 6 (Statistical Distance) *The statistical distance Δ between two discrete random variables X_1 and X_2 is defined as:*

$$\Delta(X_1, X_2) := \frac{1}{2}\sum_x(|\Pr[X_1 = x] - \Pr[X_2 = x]|)$$

Definition 7 (Distinguisher) *Given two distributions X_0 and X_1, a distinguisher, denoted $\mathcal{D}$ hereafter, is an algorithm given either X_0 or X_1, chosen randomly, which tries to guess which of the two is given.*

Concretely, we consider the following success probability

$$|\Pr[\mathcal{D}(1^k, X_0) = 1] - \Pr[\mathcal{D}(1^k, X_1) = 1]|,$$

for which we consider several cases:

1. *if the probability is equal to 0, we say that X_0 and X_1 are perfectly indistinguishable.*

2. *if the probability is lower than some value ϵ, we say that X_0 and X_1 are ϵ-statistically indistinguishable.*

3. *if the probability is negligible in k and $\mathcal{D}$ is polynomially bounded in k, we say that X_0 and X_1 are computationally indistinguishable.*

2.2.2 Pseudorandom Generator

Pseudorandom generator defines a deterministic procedure that takes as parameter a seed s (that can be random) and produces an output sequence that is indistinguishable from a truly random sequence.

2.2.3 One-way Functions

Definition 8 (One-way and Trapdoor-one-way functions) *A function $f : G \to H$ is said to be one-way if it is efficiently computable, i.e., given $x \in G$, computing $y = f(x)$ is efficient. However, inverting the function is "hard", in the sense that*

$$\forall \mathcal{A}, \Pr[a \leftarrow \mathcal{A}(1^k, b) | b = f(a)] = negl(k)$$

where $\mathcal{A}$ is polynomially bounded and the probability is taken over the randomness of $\mathcal{A}$ and the random choice of a.

If there exists some secret information s for which the invertibility of f becomes efficient, we say that f is a trapdoor-one-way function.

2.2.4 Hash Function

A hash function is an efficiently calculable function $h : \{0;1\}^* \rightarrow \{0;1\}^h$ that reduces a message of an arbitrary length to a given length that has these proprieties:

- Preimage Resistance: given $y \in \{0;1\}^h$ it is difficult to calculate x such as $f(x) = y$.
- Second Preimage Resistance: given x_1 it is difficult to find x_2 (with $x_1 \neq x_2$) such as $f(x_1) = f(x_2)$.
- Collision Resistance: it is difficult to find x_1, x_2 such that $f(x_1) = f(x_2)$.

2.2.5 Commitment Scheme

A Commitment Scheme is a protocol in which one generates a commitment value from a data that binds him to not change it but that does not reveal information about the data. The commitment then can later be opened to reveal the data. A Commitment Scheme is defined by: a message (data) $m \in M$ in the space of messages; a commitment $c \in C$ in the space of Commitments; a decommitment $d \in D$ in the space of Decommitment (that can reveal the committed data); a committing algorithm Commit and an opening algorithm Open that are used below:

- $\mathsf{Commit}(m) \rightarrow (c, d)$ the probabilistic commitment algorithm generates from the message m a commitment c and a decommitment d.
- $\mathsf{Open}(m, c, d) \rightarrow (\mathsf{true}$ or $\mathsf{false})$ the usually deterministic Open algorithm verifies if m is a valid message for the given commitment c and decommitment d.

A commitment Scheme should have the following:

- Completeness: if the Commiter and the Opener behave as specified, the opener always accepts the proof:

$$\Pr[\mathsf{Open}(m, c, d) \rightarrow \mathsf{true} | \mathsf{Commit}(m) \rightarrow \mathsf{true}] = 1$$

- Binding Property: one cannot produce two distinct messages m, m', two decommitment values d, d', one commitment value c such that $\mathsf{Open}(m, c, d) \rightarrow \mathsf{true}$ and $\mathsf{Open}(m', c, d') \rightarrow \mathsf{true}$.

  Concretely, we consider the following success probability

$$\Pr[\mathsf{Open}(m', c, d') \rightarrow \mathsf{true} | (c, d) \leftarrow \mathsf{Commit}(m), m' \neq m]$$

  for which we consider several cases:

  1. if the probability is equal to $\frac{1}{|M|}$ we say that the commitment function is perfectly binding.

2. if the probability is lower than some value ϵ we say that the commitment function is ϵ-statistically binding.
3. if the probability is negligible in k and Commit is polynomially bounded in k, we say that the commitment function is computationally binding.

- Hiding Property: For any message m the commitment c generated by $\mathsf{Commit}(m) \rightarrow (c, d)$ does not leak any information concerning m.

  Concretely, we consider the following success probability

  $$\Pr[m \leftarrow \mathcal{A}(c) | ((c, d) \leftarrow \mathsf{commit}(m))]$$

  for which we consider several cases:

  1. if the probability is equal to $\frac{1}{|M|}$ we say that the commitment function is perfectly hiding.
  2. if the probability is lower than some value ϵ we say that the commitment function is ϵ-statistically hiding.
  3. if the probability is negligible in k and Commit is polynomially bounded in k, we say that the commitment function is computationally hiding.

2.2.6 Random Oracles

Random Oracles are ideal objects that implement a uniformly distributed random function from the set X to the set Y, $\mathsf{RO} : X \rightarrow Y$ such as that for $x \in X, \mathsf{RO}(x) = y \in Y$ where y is a random value. If the same value x is given to the Oracle then the same value y will be outputted. Random Oracles are usually used to prove security in system (the system is then called *secure in the random oracle model*).

2.2.7 Digital Signatures

A digital signature scheme consist of a message $mess$ in the message space M, a signature sig in the signature space S, private key K_s,a public key K_p, a Generation algorithm Gen with some parameters $Param, k$; a signing algorithm Sign and a verification algorithm Verif that are used a follow:

- Generation: the generation of the pair of keys is done by the probabilistic function $\mathsf{Gen}(Param, 1^k) \rightarrow (K_s, K_p)$.
- Signing: the signature of the message is done by the probabilistic function $\mathsf{Sign}(mess, K_s) \rightarrow sig$.
- The verification of the signature returns true or false if $(mess, sig)$ is valid with respect to the key pair (K_s, K_p) or not with the (generally) deterministic function Verif:

  $$\mathsf{Verif}(mess, sig, K_p) \rightarrow (\mathsf{true\ or\ false})$$

Digital signatures offers the *correctness* propriety which guaranties that the Verif algorithm returns true for any the signature generated by Sign if the correct pair of keys has been used.

From this we can deduce some other proprieties:

- Authentication: gives the ability to the Verifier to check if the pretended author of the message really is the author. It also guarantees to the Signer that no one without K_s could sign the message.
- Integrity: gives the ability to the Verifier to check if the message has been tempered between the time it has been signed and verified.
- Non-repudiation: takes away the ability of the Signer to deny a genuine signature to a Verifier.
- Soundness: a valid signature cannot be proven false and an invalid signature cannot be proven true.

Traditional digital signature schemes are *universally verifiable* meaning that anyone can verify if a signature is correct or not by using the public key. With this type of signature the non-repudiation is implicitly given with the authentication and the integrity.

Security of digital signatures

When we talk about security, we generally need to distinguish between an attacker who performs a *Known Message Attack*, where the attacker can have access to messages and valid signatures form an oracle, and a *Chosen Message Attack*, where the attacker can decide of messages and get a valid signatures of the messages from the oracle.

A system that is *Chosen* message resistant offers a greater security than one that is only *Known* message resistant.

There are several different security levels to consider while talking about signatures:

- A signature is *total break resistant* if there is no way for an attacker to find the private key while only knowing the public key.
- A signature is *universal forgery resistant* if there is no way for an attacker to find a valid signature for a *random chosen* message by only knowing the public key.
- A signature is *existential forgery resistant* if there is no way for an attacker to find a valid signature for a *chosen message* by only knowing the public key.

Interactive Proof

During an interactive proof between the Prover and the Verifier, the Prover convinces the Verifier of the validity of a given statement with a proof based on a secret. An Interactive proof can have the following proprieties:

- Zero-Knowledge: the protocol does not disclose any information about the secret while being performed.
- Completeness: when the statement is true, the Verifier accepts the proof.
- Soundness: If the statement is false, the Prover cannot convince the Verifier that the statement is true.
- Non-Transferability: a third-party cannot prove a statement to the Verifier without knowing the Provers secret.

Undeniable Signatures

Undeniable signature proposed by Chaum and van Antwerpen [3] offers the ability to the Signer to keep his signature private (and therefore not universally verifiable). This type of signature scheme incapacitates anyone to verify a signature and to associate it to the Signer without his collaboration (it is called an invisible signature since one needs the private key to be able to distinguish a valid from an invalid signature). The signature verification is done by an interaction between the Signer and the Verifier, where the Signer can choose if he wants to authenticate the message and has to prove to the Verifier if the message is genuine or not. Since a Signer could very easily deny a valid signature, the signature must be undeniable so that no valid signature may be denied (for that reason in the common language we call *invisible signature* undeniable signature).

An Undeniable Signatures scheme consist of a message $mess$ in the message space M, a signature sig in the signature space S, a private key K_s, a public key K_p, a Generation algorithm Gen with some parameters $Param, k$, a signing algorithm Sign, a Confirmation interactive protocol Conf and a Denial interactive protocol Den that are used a follow:

- Generation: the generation of the keys is done by the probabilistic function Gen: $\mathsf{Gen}(Param, 1^k) \rightarrow (K_s, K_p)$.
- Signing: the signature of the message is done by the signing probabilistic function Sign: $\mathsf{Sign}(mess, K_s) \rightarrow sig$.
- The Confirmation protocol of the signature returns true if $(mess, sig)$ is *valid* with respect to the key pair (K_s, K_p) else false with the function Conf: $\mathsf{Conf}(mess, sig, K_p) \rightarrow (\mathsf{true\ or\ false})$

- The Denial protocol of the signature returns true if $(mess, sig)$ is *invalid* with respect to the key pair (K_s, K_p) else false with the function Den:Den$(mess, sig, K_p) \rightarrow$ (true or false)

Undeniable Signatures guarantees the following properties:

- Unforgeability: a valid signature can only been generated with the knowledge of the secret key.
- Soundness for confirmation: an invalid signature cannot be proven true.
- Soundness for denial: a valid signature cannot be proven false.
- Zero-Knowledge: the confirmation and the denial protocols do not disclose any information about the secret while being performed.
- Non-Transferability: a third-party cannot prove a statement to the Verifier without knowing the Provers secret.
- Invisibility: a third-party cannot distinguish a valid signature from an invalid one without knowing the K_s.

2.2.8 Diffie-Hellman

The Diffie-Hellman protocol is a key agreement protocol based on the discrete logarithm problem [6] , that has the particularity to be done over an unsecured channel without the two parties (Alice and Bob) having previously exchanged information.

The agreement works as follows: Alice and Bob need two prime numbers g and p such that g is a primitive route modulo p (g and p are not secret and can be publicly available). Alice picks a random number a and Bob a picks a random number b. Alice then generates $A = g^a \mod p$ that is sent to Bob and Bob generates $B = g^b \mod p$ that is sent to Alice. To compute the secret key k Alice computes $k = B^a \mod p = (g^b)^a \mod p$ and Bob computes $k = A^b \mod p = (g^a)^b \mod p$.

A third person who would have listened to the agreement would only know p, g, A and B which would not be sufficient to find the secret key k.

The Semi-static Diffie-Hellman protocol works exactly the same way as the DH protocol except that A is given over a authenticated channel to Bob prior the key agreement. Therefore with the Semi-static protocol, it is possible to guarantee the authenticity of Alice.

2.2.9 AES

AES is a semetric cryptography standard based on the Rijndael cipher published by Rijmen and Daemen [4]. It is block cipher based (128,192 or 256 bits) by using Substitution-permutation network on each block in 10,12 or 14 rounds (depending on the key size). Since 2001 it is an open Standard widely implemented.

2.2.10 Modes of operation

To avoid leaking information while encrypting with block cipher we have to use mode of operation. There exists several different mode of operation available (ECB, CBC, OFB, and CFB) that are more or less secure. When using ECB, someone without the knowledge of the secret key can still detect repeating blocks. The other modes use an *Initialisation Vector* (IV) and chaining between blocks to increase the security and avoid this problem.

Chapter 3

MOVA Signature Scheme

The MOVA signature Scheme has been developed by Jean Monnerat and Serge Vaudenay and presented at Asiacrypt 2004 [9]. MOVA is an undeniable signature scheme that has the particularity to generate very small signatures while keeping a good security level.

3.1 Preliminaries

We will first go through some proof protocol definitions that are used in the MOVA Scheme. The proofs and the detailed definitions can be found in [8].

3.1.1 Proof Protocol for the GHID Problem (GHIproof)

We need to define a protocol in which the Signer S can prove to the Verifier V that a set $R = (x_1, y_1), \ldots, (x_s, y_s)$ with $x_i \in X$ and $y_i \in Y$ *interpolates* in the group homomorphism $H : X \to Y$ with d= $|Y|$. The protocol could be done l times or in one batch of with l values (as done underneath).

1. V randomly picks $r_i \in_u X$ and $a_{i,j} \in_u \mathbb{Z}_d$ for $i = 1, \ldots, l$ and $j = 1, \ldots, s$. He then calculates $u_i = dr_i + a_{i,1}x_1 + \ldots + a_{i,s}x_s$ and $w_i = a_{i,1}y_1 + \ldots + a_{i,s}y_s$ for $i = 1, \ldots, l$. He then sends to S $u_1, \ldots, u_l$.

2. First, S verifies that for $i = 1, \ldots, l, H(x_i) = y_i$ (if the homomorphism is valid for all values else he aborts the protocol). Secondly, he calculates $\tilde{w}_i = H(u_i)$ for $i = 1, \ldots, l$. Thirdly S commits to the calculated values $\mathsf{Commit}(\tilde{w}_1, \ldots, \tilde{w}_l) \to (c, d)$. Finally, S sends to V the commitment com.

3. V replies with the chosen values of r_i and $a_{i,j}$ for $i = 1, \ldots, l$ and $j = 1, \ldots, s$ to S.

4. First, S verifies that the $u_i = dr_i + a_{i,1}x_1 + \ldots + a_{i,s}x_s$ for $i = 1, \ldots, l$ (if the equation is valid else he aborts the protocol). Secondly, allows V to open com by sending the $\tilde{w}_i$ for $i = 1, \ldots, l$ and de.

5. V verifies that $\tilde{w}_i = w_i$ for $i = 1, \ldots, l$ and opens the commitment with $\mathsf{Open}(\tilde{w}_1, \ldots, \tilde{w}_l, c, d)$. If the value is true the proof is valid else the proof is rejected.

3.1.2 Proof Protocol for the co-GHID Problem (co-GHIproof)

We need to define a protocol in which the Signer S can prove to the Verifier V that a set $T = ((\hat{x}_1, \tilde{y}_1), \ldots, (\hat{x}_t, \tilde{y}_t))$ with $\hat{x}_i \in X$ and $\tilde{y}_i \in Y$ *does not interpolate* in the group homomorphism $H : X \to Y$. The protocol could be done l times or in one batch of with l values (as done underneath). For the decision of non-interpolation to be taken it is enough that only iteration is proven wrong. d is the order of Y with smallest prime factor p and $R = ((x_1, y_1), \ldots, (x_s, y_s))$ with $x_i \in X$ and $y_i \in Y$ that *interpolate* in the group homomorphism $H : X \to Y$.

1. Firstly, V randomly chooses $r_{i,k} \in_u X, a_{i,j,k} \in_u \mathbb{Z}_p, \lambda_i \in_u Z_p$ with $i = 1, \ldots, l, j = 1, \ldots, s, k = 1, \ldots, t$. Secondly, V calculates for every i and every k: $u_{i,k} = dr_{i,k} + \sum_{j=1}^{s} a_{i,j,k} x_j + \lambda_i \hat{x}_k$ and $w_{i,k} = \sum_{j=1}^{s} a_{i,j,k} y_j + \lambda_i \tilde{y}_k$. (We will consider that the set $u = (u_{1,1}, \ldots, u_{l,t})$ and the set $w = (w_{1,1}, \ldots, w_{l,t})$.) Finally, V send u and w to S.

2. Firstly, S calculates for $k = 1, \ldots, t$ $H(\hat{x}_k) = \hat{y}_k$ and checks that for one or more k, $\hat{y}_k \neq \tilde{y}_k$ (if not the protocol is aboard). Secondly S calculates $H(u_{i,k}) = v_{i,k}$ for $i = 1, \ldots, l$, $k = 1, \ldots, t$. Since for at least one k, $\hat{y}_k \neq \tilde{y}_k$ and since $w_{i,k} - v_{i,k} = \lambda_i(\tilde{y}_k - \hat{y}_k)$ because of H homomorphic propriety, it is possible for S to reveal $\lambda = (\lambda_1, \ldots, \lambda_l)$. (If S does not find all the λ_i it means that V is not honest so S picks random values of λ_i). Finally V calculates $\mathsf{Commit}(\lambda) \to (c, d)$ and sends the commitment to V.

3. V replies with all the $r_{i,k}$s and $a_{i,j,k}$s.

4. S verifies that the $u_{i,k}$'s and the $w_{i,k}$'s have been calculated correctly by recalculating them with all the known values $r_{i,k}$s and $a_{i,j,k}$s. If the values are correct he replies with λ and de else he aborts the protocol.

5. V verifies that $\mathsf{Open}(\lambda, c, d) \to \mathsf{true}$ and verifies that the λ found by S are the correct ones, then accepts the proof, else he refuses it.

3.2 MOVA

To be able to use a signing scheme, we need to follow the following steps: generating the keys, signing the document, verifying the signature (acceptance and denial). Below we will develop all of them.

3.2.1 Domain Parameters

To qualify to some standards, there are some parameters $Param$ that have to be set: L_{key} would be the key length, L_{sig} would be the signature length, I_{con} would be the number iteration to confirm a valid signature and I_{den} would be the number of iterations to deny a signature. We first choose two groups X_{group} and Y_{group} (with d as the cardinality of Y_{group}) with a homomorphism h function between them (the two groups and the homomorphic function can be completely defined by their parameters: X_p, Y_p, H_p); we also need 2 pseudorandom number Generators $\mathsf{PRNG_k}$ and $\mathsf{PRNG_s}$ function (that are modelled by random oracles and defined by their seed) that takes an element from M (such as a seed $k \in M$) then generates a given number of element in the group X_{group}.

3.2.2 The keys generation scheme

The key generation is an operation that only has to be done once by the Signer from which he will produce a private key (K_s) that he will keep for himself and a public key (K_p) that he will be able to widely distribute.

1. S selects a X_{group} and a Y_{group} with an group homomorphism $h : X_{group} \rightarrow Y_{group}$ and computes the order d of Y_{group}.
2. S randomly chooses a seed k and calculates $\mathsf{PRNG_k}(k) \rightarrow (x_1, \ldots, x_{L_{key}}) = X_{gen}$.
3. S calculates $h(X_{gen}) = (y_1, \ldots, y_{L_{key}}) = Y_{gen}$.

X_{gen} and Y_{gen} are therefore subgroups of X_{group} and Y_{group}.

Our public key K_p is then $(X_P, Y_P, d, k, Param, Y_{gen})$.
Our private key k_s is then (H_p).

To guarantee security, L_{sig} must be long enough so that the probability of having an other homomorphic function that would map X_{gen} to Y_{gen} is as low as possible. In an other variant, the choice of k is done by a RA (Registration Authority) that would guarantee that a good value is chosen (the RA has also to check that the Signer is not trying to make to many registrations attempts, the RA would also needs to sign k_p as a guarantee of authenticity and the signature of k_p will be part of the public key).

3.2.3 The signing scheme

To be able to sign a message we first have to map our textual message $message$ to $mess \in M$ with a deterministic function $\mathsf{Map} : ASCII \rightarrow M$. We will need a pseudo-random generator $\mathsf{PRNG_s} : M \rightarrow X_{group}$ and a homomorphic function $h : X_{group} \rightarrow Y_{group}$.

1. $\mathsf{Map}(message) \rightarrow mess$.
2. $\mathsf{PRNG_s}(mess) \rightarrow (x_{mess_1}, \ldots, x_{mess_{L_{sig}}}) = X_{mess}$.
3. $h(X_{mess}) = (y_{mess_1}, \ldots, y_{mess_{L_{sig}}}) = Y_{mess}$

Then signed message is $(message, Y_{mess})$.

The signature is therefore $L_{sig} * log_2 d$ bits long.

3.2.4 Verification (confirmation)

In this verification phase, the Signer S proves to V the that the signature Y_{mess} is *valid* for a message $message$ and a set of keys.

1. S and V compute X_{gen} based on the K_p's value k: $PRNG_k(k) \rightarrow (x_1, \ldots, x_{L_{key}}) = X_{gen}$.
2. S and V then generate $\mathsf{Map}(message) \rightarrow mess$ and $PRNG_s(mess) \rightarrow (x_{mess_1} \ldots, x_{mess_{L_{sig}}}) = X_{mess}$.
3. The protocol finally uses the GHIproof with $R = ((X_{gen}, Y_{gen})||(X_{mess}, Y_{mess}))$.

3.2.5 Verification (denial)

In this verification phase, the Signer S proves to V that the signature Y_{mess} is *invalid* for a message $message$ and a set of keys.

1. S and V compute X_{gen} base on the K_p's value k: $PRNG_k(k) \rightarrow (x_1, \ldots, x_{L_{key}}) = X_{gen}$.
2. S and V then generate $\mathsf{Map}(message) \rightarrow mess$ and $PRNG_s(mess) \rightarrow (x_{mess_1}, \ldots, x_{mess_{L_{sig}}}) = X_{mess}$.
3. The protocol finally uses the co-GHIproof with $R = (X_{gen}, Y_{gen})$ and $T = (X_{mess}, Y_{mess})$.

Chapter 4

MOVA Train Tickets

This chapter refers to the actual implementation in Java of the MOVA protocol. The project offers a Server to generate keys and signatures, and also acts as a Prover, a Web application that requests signatures to the Server and a Terminal application that acts as a Verifier.

4.1 Structure

4.1.1 Terminology

The *Message* is the core of the train ticket containing the information about the journey that will be performed. The *Signature* is a valid MOVA Signature in respect to a given message and a key-set. The train *Ticket* is a SMS containing the Message and the Signature. The *Server* is the service that will sign the messages and then verify their authenticity. The *Terminal* is the embedded device that the train controller will have with him and that can establish a connection with the Server to verify a signature. The *WebClient* is a web-service that allows people to buy a train ticket that can establish a connection with the Server to get a message signed. The *Client* is the person who will have the SMS ticket and would present it to the train controller. The *Device* is either a WebClient or a Terminal.

4.1.2 Usual scenario

In a usual scenario, the customer will purchase his ticket on the MOVAs train ticket website, which is JPS based generated by WebClient. The WebClient will then connect to the Server in order to ask him to generate a valid signature for the given message (text of the ticket). The Client will finally receive the train ticket by SMS.

In the train, the controller will enter in the Terminal the ticket received by the Client and the Terminal will connect to the Server which will either confirm or deny the ticket.

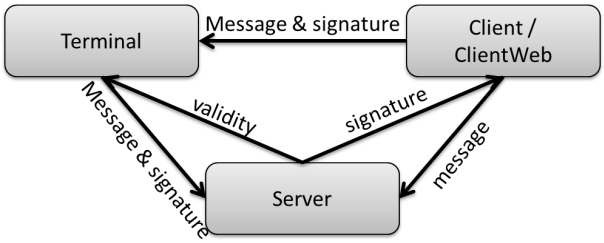


Figure 4.1: Direction of communication between the three parties

For this implementation we do not send any SMS to the client but for a real deployment we could use a SMS service provider such as TrueSenses (`www.truesenses.com`) to receive tickets request by SMS and then to send the Tickets to the client's cell phone by SMS.

4.2 Implementation

The implementation is done in 4 different java projects. To use them, one should have JRE (`http://www.java.com/en/download/manual.jsp`) and a tomcat server (`http://tomcat.apache.org`) installed and running on the computer.

- **the Server Project** gives the Server service to the environment. It takes as argument the port number it has to listen to (if no port specified it will use the port 5000). Usually only one server should be launched per environment on a powerful computer so it can handle many simultaneous connections.
- **The Terminal Project** gives a Terminal service to the environment. It takes as argument the IP address and port that it should use to connect to the Server (if not specified, it will use localhost and port 5000). The Terminal project should be launched on embedded devices with data connection. We can imagine that every train controller would have this kind of device with him.
- **The ClientWeb Project** gives the WebClient service to the environment. At launch the user (it would be the administrator of the web site) must enter the IP address and the port to connect to the Server. The ClientWeb Project can be launched at any MOVA Train Ticket partner (such as Swiss or CFF) so they can offer the service to their clients.
- **The MovaShared Project** is not a runnable project but contains all the shared classes of the three other ones. All the classes of MovaShared are already included in the other projects.

If the Terminal Project or the ClientWeb Project are launched before the Server project, they won't be able to connect and will have to be rerun to connect.

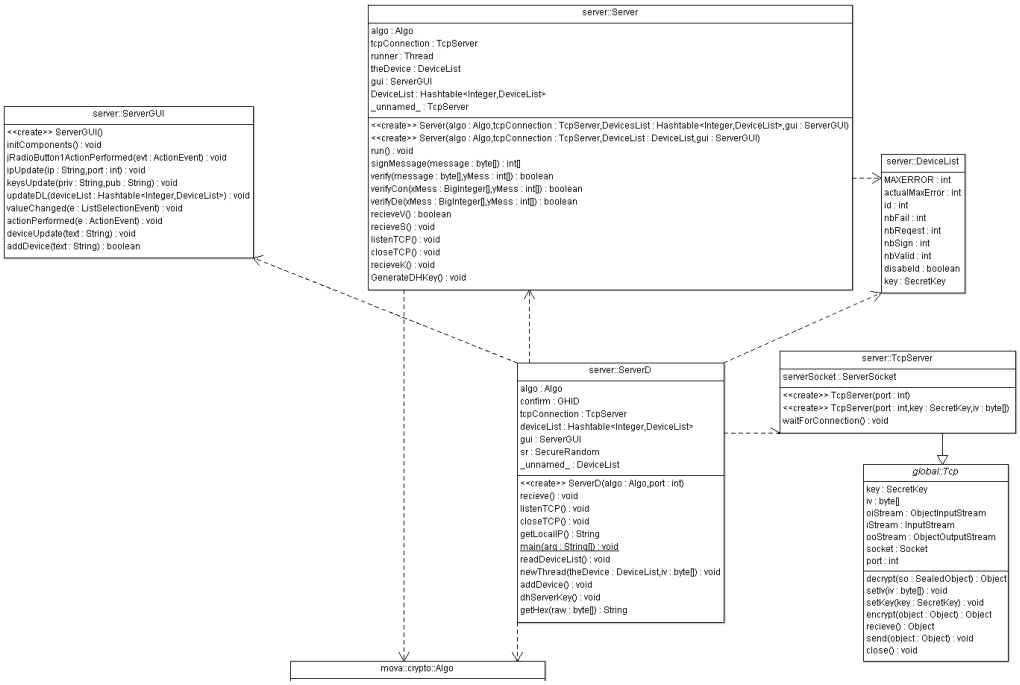


Figure 4.2: UML of the Server Package

4.2.1 Server

At launch the Server will first generate its MOVA signing keys (or load the ones saved in the files *keyPriv.key* and *keyPub.key*) and load the list of all the known authorized devices (the list is in the file *DeviceList.list* and points to other files with the complete information about specific devices). Then Server runs as a demon (*ServerD.class*) always waiting for incoming connections on his listening port (by default 5000). When an incoming connection arrives, the Server verifies if the Device is in the DeviceList (with its unique ID): if the answer is positive, the Server replies with an open port number handled by *Server.class* in a new thread; if the answer is negative, the Server requests from the Administrator an approval to add the new Device, creates a DeviceList entry, agrees on a AES secret key and sends back the unique ID of the Device.

Once connected to the new thread, the Server sends the IV for the AES encryption (after that, all the following communications are encrypted), then the Device can request from the Server to send his public key, sign a message or verify a message. When the interaction is finished, the thread connection is closed and the thread is killed.

At launch of the Server, there is a User Interface (*ServerGUI.class*) window that shows the current Devices IDs with a short history of the interaction with all of them, a log of the actions of the Server and the ability to ban a specific Device

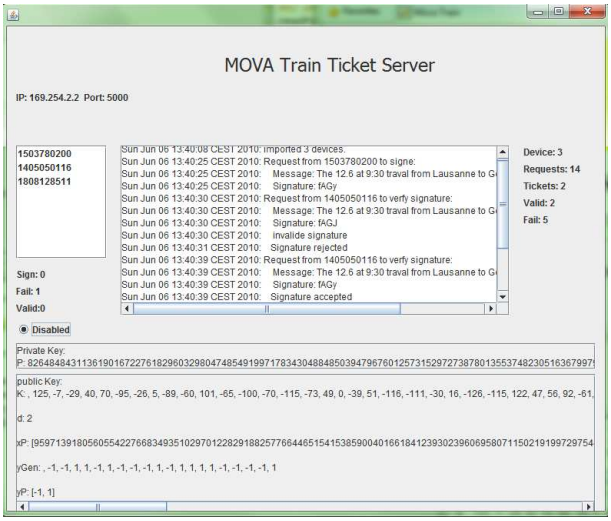


Figure 4.3: Screen shot of the Server's UI

(in case one of them is stolen or cheating). When the Server starts for the first time, he generates the private and public key that are saved in the *keyPriv.key* and *KeyPub.key* files, so in case of crash or reboot from the Server the same key set can be used. If the Server for any reason wants to change his key set, the administrator must shut-down the Server, delete the .key files and relaunch the Server.

In order to to keep track of all the Devices that are authorised to communicate with the Server, the Server keeps a Hashtable of DeviceList (*TerminalList.class*) that contains the unique ID of a Device, the number and type of requests made and its status. Every DeviceList's ID is saved into a the *deviceList.list* file and points to other files containing the DeviceList data and is read at every launch of the Server. If the Server for any reason wants to remove the devices, the administrator must shut-down the Server, delete the *deviceList.list* file and all the files with the *dl* extension, then relaunch the Server. To avoid too many attempts with false signatures by a Device, after a given number of failed tries (10 for demonstration purposes) the Device is banned and the administrator must rehabilitate it.

4.2.2 Terminal

The Terminal (*Terminal.class*) can only do some very simple tasks, it is mainly here as an interface to the access the Server's services and keep some personal statistics. At launch it will request to the Server to receive an ID which will be used to authenticate at every interaction and they will agree with the DH protocol on a AES key. The user interface (*TerminalGUI.class*) is intuitive and effective: it is constituted by two textboxes, one for the message and one for the signature, and a verify button which initiates the verification protocol. Once launched, the interface will display *Valid* in green if the signature matches the message or display *Invalid*

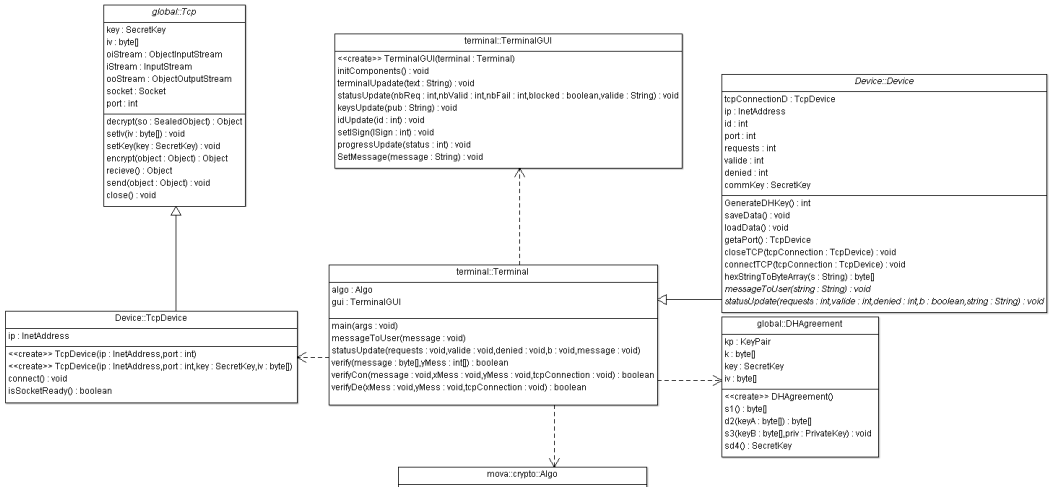


Figure 4.4: UML of the Terminal Package

in red if the message and the signature do not match.

4.2.3 ClientWeb

The ClientWeb runs a singleton instance of the signing service (*SignService.class*) that generates objects and text for the JSP pages, and creates an instance of the client service (*Client.class*) that will handle the communication with the Server. At launch the client service will request from the Server to receive an ID which will be used to authenticate at every interaction and they will agree with the DH protocol on a AES key. The different lists from the drop down-menu (*OptionList.class*) enable the User to generate the text from pre-existing information or the User (for demonstration purpose) can choose the text he wants (in a real deployment they should be a database behind the data for the drop-down list).

4.3 Usage

To use the MOVA Train Ticket, one must first launch the Server, which will generate in the *Server.jar* folder 4 files with the *key* extension. The *DhKpPub.key* is the DH public key, used for the Semi-static DH key-agreement. This file must be shared with the Terminal and the ClientWeb (it only has to be done the first time). To do so, one must insert it in the *Terminal.jar* file and in the *WEB-INF/classes* folder of the *ClientWeb.war* file. (To edit Jar and War files easily one can use 7Zip `http://www.7-zip.org`). Then one has to deploy the *ClientWeb.war* to the Tomcat Server, and launch the *Terminal.jar* on a device.

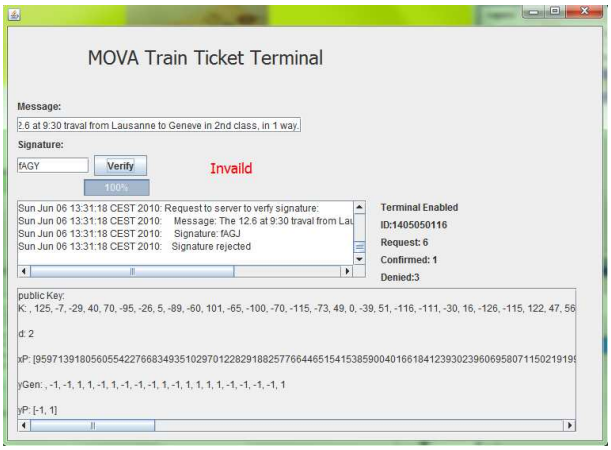


Figure 4.5: Screen shot of the terminal's UI

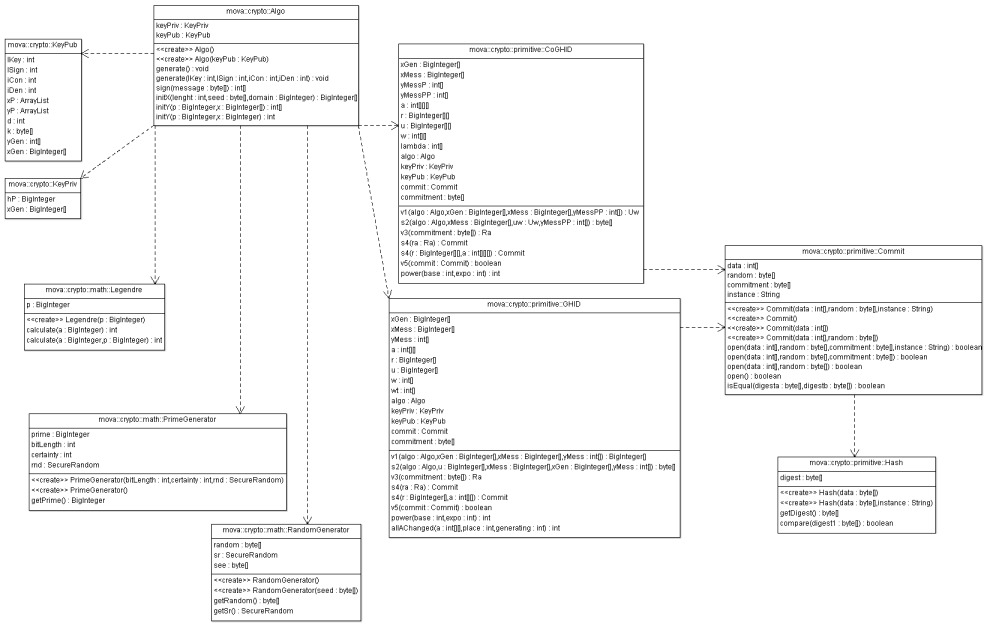


Figure 4.6: The UML of the MOVA algorithm used by Server and Terminal

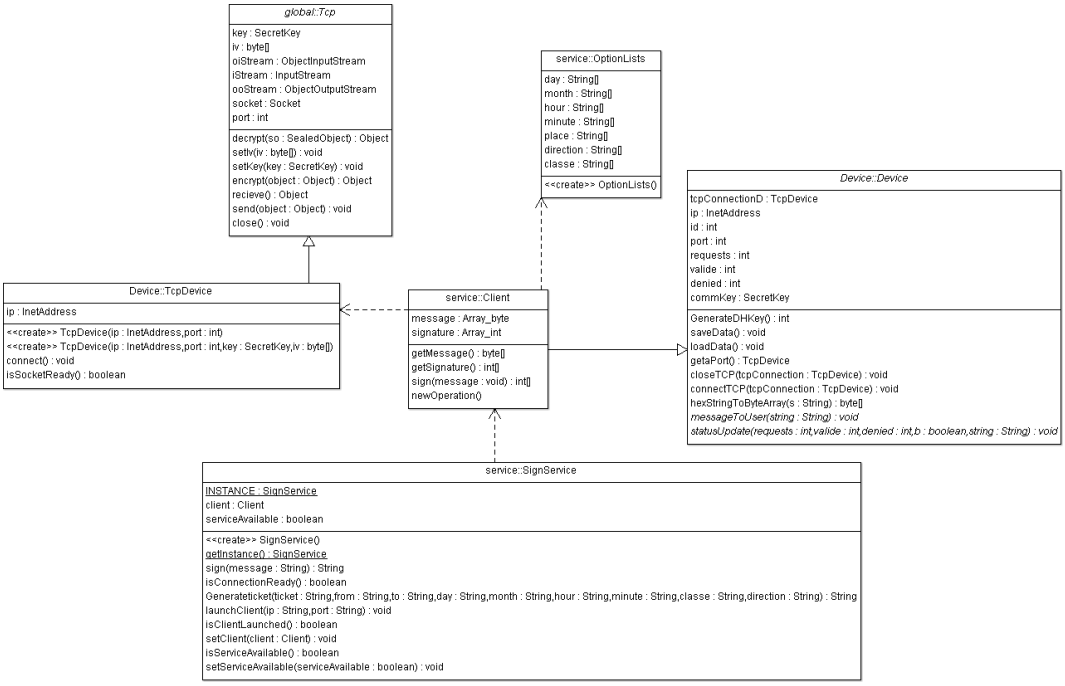


Figure 4.7: UML of the ClientWeb Package

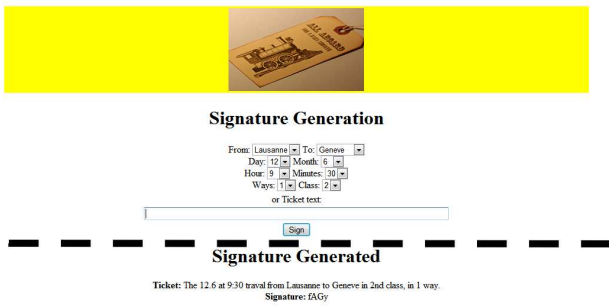


Figure 4.8: Screen shot of the ClientWeb's UI

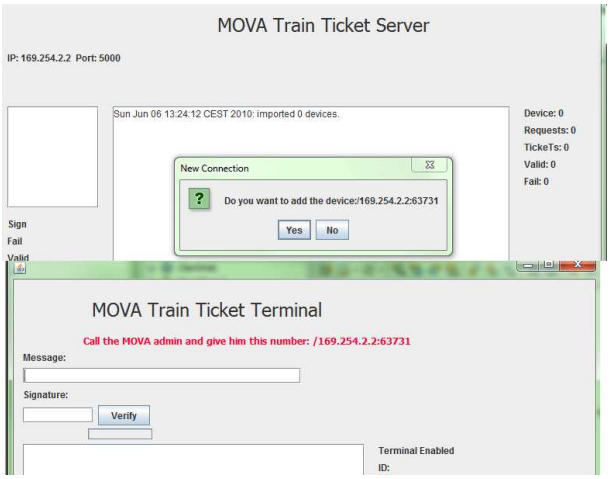


Figure 4.9: Request for approval to add a Device

4.4 Communication Protocols

4.4.1 General

All the communications are initiated by a Device (Terminal or ClientWeb). The Device sends his ID (-1 if he has no ID yet) in clear, then the Server replies with the AES IV and the port number on which the communication will then take place. The following communications all happen in a new Thread on the port number previously sent:

4.4.2 Diffie-Hellman

At the first launch of the Server, it generates its private a and public $A = g^a \bmod p$ keys and saves them to a file (*dhKpPub.key*). (The Administrator must then put this file in all the devices.) Since the Device has no ID, the *GenerateDHKey()* method in the *Server.class* and *Device.class* are called.

Client: first the Client loads the Server's public key (*dhKpPub.key* and *dhKpPriv.key*), secondly generates his own key-set, thirdly sends his public key to the Server, fourthly computes the secret key based on his private and the Server's public key, finally derives from the secret key an AES key and the IV.

Server: first the Server loads his private key (*dhKpPriv.key*), secondly receives the public key of the device, thirdly computes the secret key based on his private and the device's public key, finally derives from the secret key an AES key and the IV. The Client and the device have now a shared AES key, and the Server send the ID of the device (in an encrypted way).

In all the future communications, the IV will be generated by a separated Secure random generator.

4.4.3 Public Key

This protocol is only between the Terminal and the Server. All the communications are encrypted. The Terminal calls the method *getkey()* then sends the *k* command to the Server. The Server then replies with his public MOVA key that is then displayed on the Terminal's UI.

4.4.4 Sign

This protocol is only between the ClientWeb and the Server. All the communications are encrypted. The ClientWeb calls the method *Client.sign(byte[] message)* with the message as parameter, then sends the *s* command followed by the message. The Server then replies with the signature (in boolean) to the ClientWeb (the signature will been shown to the user in ASCII caracter).

4.4.5 Verify

The protocol is only between the Terminal and the Server. All the communications are encrypted. The Terminal calls the method *verify(byte[] message, String yMess)* with the message and the signature as parameter, then sends the *v* command, the message and the the signature (converted to boolean) to the Server; then the Confirm protocol (GHID) starts. If at any stage of the protocol an error occurs, the message null is sent, and then the deny protocol (co-GHID) starts, otherwise the protocol finishes correctly.

Confirm

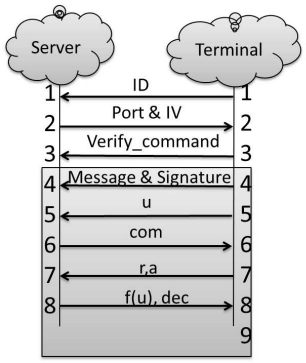
The confirm protocol works as described in the previous chapter. If at any stage the message received or the result return are not as expected, the deny protocol starts. If the protocol successfully finishes a *Valid* massage appears on the Terminal and in the log of the Server. (The w_i values chosen by the Terminal and found by the Server are printed in the java terminal).

Deny

The deny protocol works as described in the previous chapter. If at any stage the message received or the result return are not as expected, the machine is marked as cheating and is banned by the Server. If the protocol successfully finishes a *Invalid* message appears on the Terminal and in the log of the Server. (The λ_i values chosen by the Terminal and found by the Server are printed in the java terminal).

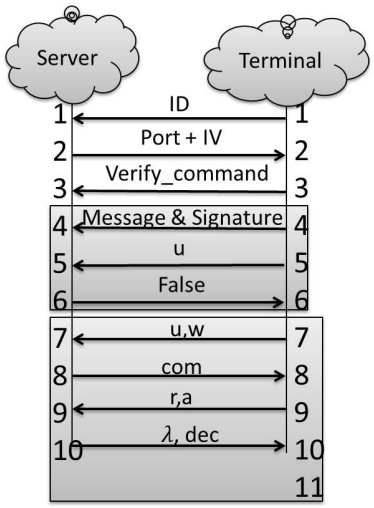
4.5 Implementation

In this section we are going to see more into details how the methods have been implemented and which settings have been used.



1	size: 81 B
2	size: 201 B
3	size: 50 B
4	size: 187 +107 = 294 B
5	$r \in_u G, a_j \in_u \mathbb{Z}_d$ calculates :$u = dr + a_1 g_1 + \cdots + a_s g_s$ $v = a_1 e_1 + \cdots + a_s e_s$
	time: 58 ms
	size: 3'509 B
6	calculates: $(com, dec) \leftarrow Commit(f(u))$
	time 127 ms
	size: 59 B
7	size: 4'835 B
8	calculates :$u = dr + a_1 g_1 + \cdots + a_s g_s$
	time: 13 ms
	size: 431 B
9	Verifies that $f(u) = v$ and $Open(v, com, dec) \rightarrow 1$
Time:	**199 ms**
size:	**9'460 B**

Figure 4.10: GHID protocol



1	size: 81 B
2	size: 201 B
3	size: 50 B
4	size: 187 +107 = 294 B
5	$r \in_u G, a_j \in_u \mathbb{Z}_d$ calculates :$u = dr + a_1 g_1 + \cdots + a_s g_s$ $v = a_1 e_1 + \cdots + a_s e_s$
	time: 58 ms
	size: 3'509 B
6	calculates: $verify(mess, sign) \rightarrow false$
	Time: 64 ms
	size: 47 B
7	$r_k \in_u G, a_{j,k} \in_u \mathbb{Z}_d, \lambda \in_u \mathbb{Z}_p$ $u_k = dr_k + \sum_{j=1}^{s} a_{j,k} g_j + \lambda x_k$ $w_k = \sum_{j=1}^{s} a_{j,k} e_j + \lambda z_k$
	Time: 174 ms
	Size:68'360 B
8	$v_k = f(u_k), y_k = f(x_k)$ Find λ form $w_k - v_k = \lambda(z_k - y_k)$ $Commit(\lambda) \rightarrow (com, dec)$
	Time:127 ms
	Size 59 B
9	Size:102'822 B
10	$u_k = dr_k + \sum_{j=1}^{s} a_{j,k} g_j + \lambda x_k$ $w_k = \sum_{j=1}^{s} a_{j,k} e_j + \lambda z_k$
	Time: 168 ms
	Size: 431 B
11	Verifies that λ is correct and $Open(\lambda, com, dec) \rightarrow 1$
Time:	**591 ms**
Size:	**175'854 B**

Figure 4.11: co-GHID protocol

4.5.1 MOVA

Homomorphism function

In the MOVA original paper several homomorphism functions have been proposed such as character on $\mathbb{Z}_n^*$ or RSA, each giving different proprieties and efficiencies. In this project the priority has been put on efficiency by using the Legendre symbol $\left(\frac{a}{p}\right)$, (with $n = pq$ and p,q prime) which is defined as follow:

$$\forall a \in \mathbb{Z}_p^\star \left(\frac{a}{p}\right) = \begin{cases} 1 & \text{if } a \text{ is a quadratic residue modulo } p \\ -1 & \text{if } a \text{ is a quadratic nonresidue modulo } p \end{cases}$$

The Xgroup and Ygroup

Since we have chosen to use the Legendre symbol as homomorphism function the *Ygroup* = {-1,1}. In the internal use of the application and while sending messages, the *Ygroup* is represented as a boolean mapping $1 \rightarrow \mathsf{true}$ and $-1 \rightarrow \mathsf{false}$. When the signature must be sent in an SMS, it is then converted into ASCII characters. To avoid confusing some characters (such as 'o', 'O', '0') have been removed, so the mapping will only be done by converting 5 bits into 32 different characters. (all the methods are in *ConvertY.class*).

The *Xgroup* are all the elements in $\mathbb{Z}_n^*$. We will generate two uniformly random prime numbers of 512 bit (p and q that will be kept secret,) to then calculate n ($= pq$). It is computationally hard to calculate the symbol $\left(\frac{a}{p}\right)$ by only knowing n since it implies having to find the factorization of n.

Security parameters

Since we only take 5 bits to generate a ASCII character, it is important to use a small signature. In the MOVA original paper there is proof that, given a 20 bit signature:

- A Prover must have in his possession the secret key in order to interactively validate or deny a signature with a probability larger than 2^{-20}.
- If the probability of forging a valid signature is larger than 2^{-20} then the forger must have in his possession the secret key.
- If the Advantage of a Distinguisher between a valid and invalid message is larger than 0 then the Distinguisher has the secret key.

In this implementation, *iCon* and *iDen* are set to 20 which means that for a signature to be accepted or rejected it has to pass the GHID or coGHID protocol 20 times (this represents a probability of 2^{-20} to pass the protocol when it should not).

4.5.2 Hash

The Hash function used in this project comes from *Hash.class*. It is based on the Java security library (*java.security.MessageDigest*), uses *SHA-256* and for any given message, it outputs a 32 bytes digest.

4.5.3 Commitment

The commitment function in this project comes from *Commit.class*. It takes as input a message, concatenates it with a 128 random bytes then does a hash function on it that outputs a digest of 32 bytes (256 bits).

The *binding property* is given by the collision resistance of our hash function. With the birthday paradox, an attacker needs to compute $2^{n/2}$ (2^{128}) hash operations to likely find a collision.

The *hiding propriety* is given by the first pre-image residence of our hash function. An attacker needs to compute 2^n (2^{256}) hash operations to likely find a valid pre-image.

4.5.4 Legendre

The Legendre symbol is calculated in the *Legendre.class* file. The method to do the calculation is (*calculate(BigInteger a, BigInteger p)*) and applies this formula:

$$x \leftarrow a^{\frac{p-1}{2}} \mod p$$

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & x = 1 \\ -1 & x = -1 \end{cases}$$

4.5.5 PseudoRandom Generator

The Pseudo-random generator and the pseudo-random data are handled in *RandomGenerator.class* which instantiates a *SecureRandom* object (part of package java.security package). It uses a deterministic function with a random seed ;it can either self-seed getting it from the OS (usually using the timing of I/O events) [1] or use a seed given in parameter. This implementation uses the java's recommended function *SHA1PRNG* which has a period of 2^{160}.

4.5.6 Diffi-Hellman

The Diffi-Hellman implementation is done in *DHAgreement.class* following the Semi-static method. The public key of the Server, generated by the method *s1()* is saved into a file and installed in all the devices. The key used to make the agreement is 1024 bits long.

The methods *s1() and s3(byte[], PrivateKey)* are called by the Server and the method *d2()* is called by the devices.

The implementation is inspired by [10] proposal.

4.5.7 AES

The AES implementation is done in *DHAgreement.class* after the secret key has been generated. The secret key is used as a seed for a *SecureRandom* that will generate the AES key (of 128 bits for portability reasons). The messages are then encrypted /decrypted in the *Tcp.class* using the AES key generated in the DH agreement and and the IV set by the Server. The AES is used with CBC mode of operation and encrypts *serialized object* into *SealedObject*.

Chapter 5

Conclusion

With this work, we have first seen some mathematical and cryptographic background to understand the MOVA scheme, then we have studied the MOVA scheme and identified its special properties that make this project possible. After all this being set, we have proposed a real implementation of the MOVA train Ticket service.

The Java implementation can prove the viability of such a system and shows that it can be done with an easy interface to generate, control and administrate train tickets and related devices.

There might still be some issues to real-world realisation that are not related to the MOVA security, such as assuring the unicity of a message and that passengers do not transmit tickets to each other; and the need of permanent data connectivity for the controller to always be able to verify tickets.

A future work could give deeper look into these problems.

Bibliography

[1] Randomness Recommendations for Security, RfC-1750, 1994.

[2] F. Chabaud and S. Vaudenay. Links between differential and linear cryptanalysis. In *Advances in CryptologyEUROCRYPT'94*, page 356. Springer, 1995.

[3] D. Chaum and H. van Antwerpen. Undeniable signatures, Advances in Cryptology-Crypto 1989, LNCS 435, 1989.

[4] J. Daemen and V. Rijmen. AES proposal: Rijndael. 1999.

[5] O. Goldreich. *Foundations of cryptography: Basic applications*. Cambridge Univ Pr, 2004.

[6] M. Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[7] J. Knudsen. *Java cryptography*. O'Reilly Media, Inc., 1998.

[8] J. Monnerat. Short Undeniable Signatures: Design. *Analysis, and Applications. PhD thesis*, 3691, 2006.

[9] J. Monnerat and S. Vaudenay. Generic homomorphic undeniable signatures. *Advances in Cryptology-ASIACRYPT 2004*, pages 1–6.

[10] S. Oaks. JAVA Security, Ed, 2001.

[11] S. Vaudenay. *A classical introduction to cryptography: applications for communications security*. Springer-Verlag New York Inc, 2005.